Building Reusable Repertoires for Stochastic Self-* Planners

Cody Kinneer
School of Computer Science
Carnegie Mellon University
Pittsburgh, USA
ckinneer@cs.cmu.edu

Rijnard van Tonder
School of Computer Science
Carnegie Mellon University
Pittsburgh, USA
rvantond@alumni.cmu.edu

David Garlan
School of Computer Science
Carnegie Mellon University
Pittsburgh, USA
garlan@cs.cmu.edu

Claire Le Goues
School of Computer Science
Carnegie Mellon University
Pittsburgh, USA
clegoues@cs.cmu.edu

***Abstract*—Plan reuse is a promising approach for enabling self-* systems to effectively adapt to unexpected changes, such as evolving existing adaptation strategies after an unexpected change using stochastic search. An ideal self-* planner should be able to reuse repertoires of adaptation strategies, but this is challenging due to the evaluation overhead. For effective reuse, a repertoire should be both (a) likely to generalize to future situations, and (b) cost effective to evaluate. In this work, we present an approach inspired by chaos engineering for generating a diverse set of adaptation strategies to reuse, and we explore two analysis approaches based on clone detection and syntactic transformation for constructing repertoires of adaptation strategies that are likely to be amenable to reuse in stochastic search self-* planners. An evaluation of the proposed approaches on a simulated system inspired by Amazon Web Services shows planning effectiveness improved by up to 20% and reveals tradeoffs in planning timeliness and optimality.**

***Index Terms*—self-*, planning, search-based, genetic programming, repertoires**

I. Introduction

The increasing size and complexity of software systems motivates self-adaptation, to allow systems to operate in environments with uncertainty. Self-* approaches have been successful in enabling systems to grapple with changing environments [1]–[3]. This self-* automation often relies on a *planner*, which determines the appropriate adaptation tactics for the system to use in response to change, arranged in an adaptation strategy or *plan*. Whether online [4] or offline [5], planners facilitate adaptation by making decisions based on the capabilities of the system, the environment, and the system's quality objectives like cost and latency, including making tradeoffs between competing objectives.

While self-* techniques can allow systems to adapt to changes considered at design time, they often struggle to handle unforeseen changes, those changes not considered at design time. Such changes can violate assumptions that the system was designed on, resulting in the system failing to achieve its objectives. Examples of these changes include the addition or removal of adaptation tactics, changes in the effects of these tactics, or changes to the quality objectives. When such changes occur, the self-* planner must *replan*. This is expensive and resource-intensive, whether the plans are human-written or automatically produced, especially as self-* systems grow in size and complexity.

One promising solution to this problem is to leverage existing planning knowledge instead of replanning from scratch. That is, plan *reuse* may allow these systems to incrementally evolve in response to unexpected changes. We envision an ideal self-* planner that can effectively reuse a repertoire of existing adaptation strategies. However, while intuitive, plan reuse is difficult [6] and must be applied thoughtfully to result in a positive outcome [7]. Of particular concern is the large evaluation overhead associated with evaluating the applicability of the existing plans, which can quickly outweigh the benefits of reuse. This means that to benefit from the knowledge encoded in the planner's repertoire, a repertoire of prior knowledge must be amenable to reuse in the following key ways: (a) the plans in the repertoire should be likely to generalize to future situations, and (b) the plans in the repertoire should be cost effective to evaluate.

We propose and evaluate a novel, two part approach for constructing effective reusable repertoires. First, we take inspiration from chaos engineering [8] to explore the change space by randomly generating change scenarios and corresponding adaptation strategies to build a base of planning knowledge. In the second phase, we use the insight that adaptation strategies are similar to software programs to apply program analysis approaches to the base of adaptation strategies, with the aim of extracting planning components that are likely to generalize and are cost effective to evaluate.

We evaluate the proposed approach for building reusable repertoires by presenting a stochastic self-* planner (extended from our prior work [7], which could only reuse a single strategy) that reuses a repertoire of adaptation strategies to replan after unexpected changes. We show the resulting planner is more robust and responsive to a a broader range of unexpected change scenarios than prior work. We evaluate the proposed approach using an exemplar self-* system, a cloud-based web server inspired by AWS, and show that replanning effectiveness improved by up to 20%, as well as reveal trade

This work is supported in part by award N00014172899 from the Office of Naval Research. This material is based upon work supported by the NSA under Award No. H9823018D0008. This research supported in part by the National Science Foundation (CCF-1750116, CCF-1618220). Any views, opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsoring agencies.

offs in planning optimality and timeliness.

The key contributions of this paper are as follows:

1) A two step approach for generating reusable repertoires for stochastic self-* planners to more effectively replan.
2) A technique, inspired by chaos engineering, for exploring the change space of self-* systems to facilitate the construction of reusable repertoires.
3) An approach for identifying reusable plan fragments based on software source code clone detection.
4) A rule-based approach including a collection of syntactic transformation rules for AST based planning languages to extract reusable planning components.
5) An empirical investigation of building reusable repertoires for a cloud-based web server inspired by Amazon Web Services, over a wide range of automatically generated unexpected change scenarios.

Section II provides background, and introduces our exemplar system. Section III describes our approach for generating reusable repertoires for self-* systems. Section IV describes the results of our evaluation. Section V positions the paper with respect to related work, and Section VI concludes.

II. Background and Exemplar

Self-* systems are software-centric systems that automatically take action in response to changes in their environments in order to continue satisfying their quality attribute requirements. Frequently, these systems are arranged according to the well-known MAPE-K architecture [9]. The architecture typically consists of two layers, a *managed* system, and a five-component *managing* system. The managing system gathers information on the state of the managed system and its environment using sensors, and can take actions that affect the managed system via actuators. The monitor component gathers information from the sensors, which the analyze component examines to determine when adaptation is necessary. The plan component then decides which adaptation tactics the system should use to adapt, and the execute component carries out the plan using the actuators. The last component K provides a shared store of information for the other components to use.

In this work, we focus on the planning component, and propose a novel planning approach that effectively generates and then reuses knowledge to help systems respond to unanticipated changes. The rest of this section provides background on planning, and the planner we extend from prior work (Section II-A); and outlines the exemplar self-* system we use to explain and evaluate our work (Section II-B).

A. Planning with reuse and stochastic search

The planner is a key component in a self-* systems since it is responsible for determining how the system should adapt. The planner outputs an adaptation strategy or plan, which consists of an ordered collection of tactics. There are several approaches for planning, including online planners [4] which generate plans at runtime, and offline planners [5] that produce strategies offline which are then chosen at runtime; these two

⟨*plan*⟩ ::= ‘(’ ⟨*operator*⟩ ‘)’ | ‘(’ ⟨*tactic*⟩ ‘)’

⟨*operator*⟩ ::= ‘`F`’ ⟨*int*⟩ ⟨*plan*⟩ (For loop)
| ‘`T`’ ⟨*plan*⟩ ⟨*plan*⟩ ⟨*plan*⟩ (Try-catch)
| ‘`;`’ ⟨*plan*⟩ ⟨*plan*⟩ (Sequence)

⟨*tactic*⟩ ::= ‘`StartServer`’ ⟨*srv*⟩ | ‘`ShutdownServer`’ ⟨*srv*⟩
| ‘`IncreaseTraffic`’ ⟨*srv*⟩ | ‘`DecreaseTraffic`’ ⟨*srv*⟩
| ‘`IncreaseDimmer`’ ⟨*srv*⟩ | ‘`DecreaseDimmer`’ ⟨*srv*⟩

Fig. 1: Grammar for specifying plans. Servers (*srv*) can be any of the 16 availability zones listed in Table I; For loops can iterate up to 10 times.

paradigms tradeoff between plan optimality, and time. Existing planners generally struggle to handle unanticipated scenarios.

We instantiate our proposed approach for reusable repertoires by extending a planner we proposed in our prior work [7], which investigated plan reuse using genetic programming. Genetic programming [10] (GP) is a population-based stochastic search approach inspired by evolutionary processes in biology, and is a type of genetic algorithm [11]. In our context, the search traverses a space of possible plans, written in a simple domain-specific planning language inspired by `Stitch` [5], and then represented as abstract syntax trees (ASTs). The genetic program evaluates the fitness of the candidate plans by simulation. Figure 1 shows the grammar. Individual tactics can be composed using sequencing (the `;` operator), loops (the `F` operator), or a try-catch operator (the `T` operator). The tactics expressed in the language correspond to the atomic adaptation tactics available to the self-* system, and is therefore specific to the considered system; we describe our exemplar system next.

Our prior approach was predicated on the idea of *reuse*: when an unexpected change occurs, the search-based planner replans, using a known-good previous plan to seed the search population. However, effective plan reuse is difficult [6], and our previous approach required the development of reuse-enabling techniques to support it. In this work, we observe that plans can be treated as programs and analyzed accordingly to help identify semantically useful components for reuse; we therefore present a self-* planning approach that develops and then reuses carefully-constructed repertoires of adaptation strategies.

B. Exemplar system

Our exemplar system is a cloud-based web server running on an infrastructure inspired by Amazon Web Services (AWS), which has been built based on the SWIM [12] examplar to evaluate other planning approaches [4], [7]. The goal of the system is to serve content in response to user requests. The system should perform this function in a way that maximizes several different (and competing) quality attribute requirements, and the system has access to several adaptation tactics to accomplish this.

Location	Name	Cost in $ per instance per month	Number of Availability Zones
N. Viriginia	us-east-1	69.12	6
Ohio	us-east-2	69.12	3
Oregon	us-west-2	69.12	4
Mumbai	ap-south-1	72.72	3
Stockholm	eu-north-1	73.44	3
Canada	ca-central-1	77.04	2
Ireland	eu-west-1	77.04	3
London	eu-west-2	79.92	3
Paris	eu-west-3	80.64	3
N. California	us-west-1	80.64	2
Frankfurt	eu-central-1	82.80	3
Seoul	ap-northeast-2	84.96	3
Singapore	ap-southeast-1	86.40	3
Sydney	ap-southeast-2	86.40	3
Tokyo	ap-northeast-1	89.28	3
Sao Paulo	sa-east-1	110.16	3

TABLE I: Regions in exemplar system with cost and number of availability zones. Cost data from Concurrency Labs [13].

Architecture. The architecture of the cloud service provider allows the system to provision virtual server instances. These instances may be requested based on availability zones, which provide a way to provision instances on architecturally separate pieces of infrastructure (i.e., a failure in one availability zone should be contained to that zone, and instances running in other zones are expected to remain available). These availability zones are grouped based on a higher-level architectural entity called regions, which provide additional isolation for reliability purposes. Table I shows the regions available in the exemplar system, along with the number of availability zones in each region, and the cost of starting up a server instance in that region. In total, there are 50 availability zones spread across 16 regions. The considered regions and number of availability zones per region are both based on AWS. The cost per month information was obtained from Concurrency Labs based on AWS's Price List API [13] for a c5.large instance. For the purposes of the exemplar, we assume that the system can only utilize this instance type.

Quality attributes. The system's quality attributes are profit and user-experienced latency. To generate profit, the system can serve advertisements along with the users' requests, but profit is reduced by the costs of running server instances. The system has the ability to not serve high definition images and media content (including ads) to speed up handling of requests, at the cost of reduced quality and ad revenue. The latency quality attribute is the amount of time users spend waiting for their request to be served, and can be measured as the number of users that need to wait longer than an acceptable threshold. Since these quality attributes are conflicting, the system must take care to balance them appropriately.

Uncertainty. Complicating balancing these quality attribute requirements are several sources of uncertainty that the system must manage. One source of uncertainty is the number of users sending requests to the system, which can change. Additionally, the reliability of the underlying cloud infrastructure is questionable, e.g., server instances can fail, or the available server instance characteristics can change.

Adaptation tactics. To manage the uncertainty in the environment, the system has several adaptation tactics that can be used to respond to changes in the environment. These tactics are to start or shutdown instances, raise or lower a dimmer, and to adjust the proportion of requests directed to each availability zone. The system can start or stop server instances on a per availability zone basis. For the purposes of the exemplar system, we will assume that a maximum of 5 instances can be running at a time at each availability zone. The dimmer controls the proportion of requests that are served with low-fidelity content (and without ads). A higher dimmer value allows the system to respond to more requests in the same amount of time, but reduces content quality and system revenue. The dimmer can be changed in 25% intervals and can be set on a per availability zone basis. The system can adjust traffic allocation by changing a traffic level parameter at each zone. This parameter can be a value between $0 - 4$, and traffic is allocated to each zone proportionally. Since there are 50 availability zones, each with 5 settings for the dimmer value, traffic value, and 6 settings for the number of instances, there are 6×10^{108} configurations.

The behavior of server instances depends on three attributes, cost, power, and brownout ratio. The cost is the amount of money charged by the service provider to run an instance per unit time. The power represents how many dimmed requests (low-fidelity content and without ads) can be served per unit time. The brownout ratio is the ratio of dimmed requests to full requests that can be served (e.g., an instance with a brownout ratio of 2 could serve twice as many dimmed requests as full requests for the same unit of time). These attributes are set on a per availability zone basis, and by default, the costs are set according to regions as shown in Table I. While the power and brownout ratio are set to 1000 dimmed requests per second and two respectively, for all availability zones.

Change scenarios. To study how self-* systems respond to various types of unexpected changes, the exemplar system supports the easy generation of change scenarios. A change scenario is defined as a vector of attributes that influence the system's behavior and utility. There are a total of 159 attributes that can be changed. These consist of three attributes that apply to the system as a whole, including the number of incoming requests, and coefficients on the profit and latency values (these coefficients control the weighting of these conflicting quality attributes in the system's utility function). Six values determine the tactic failure rates of each of the six tactics available to the system. The remaining attributes are the instance cost, power, and brownout ratio, which can be manipulated (to obtain new change scenarios, not by the system) for each of the 50 availability zones, resulting in 159 attributes total.

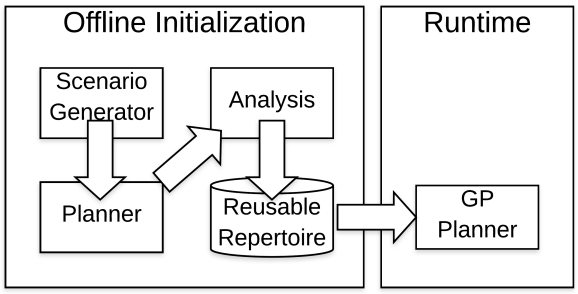

Fig. 2: A high level view of the approach.

III. Approach

We introduce a planner that aims to effectively reuse prior knowledge. However, as we learned in our previous work [7], using individual prior plans to seed a replanning effort is not always satisfactory, almost by definition: individual plans do not account for unanticipated changes. We therefore propose to build *repertoires* of useful prior knowledge to seed replanning.

Figure 2 overviews the approach, which divides the planning process into an offline and runtime step. During the offline initialization phase, we construct a reusable repertoire of adaptation strategies for the planner to incrementally evolve at runtime. This phase is further subdivided into a two step process: firstly exploring the space of randomly generated change scenarios and producing adaptation strategies to address them, and then analysing the generated adaptation strategies to extract generalizable and cost effective components for the repertoire. In the online phase, we extend our prior genetic programming planner [7] by seeding it with the adaptation strategies in the repertoire.

A key idea behind repertoire construction is that certain "pieces" of plans are particularly informative for reuse. For example, repeated planning components, such as starting more instances of the most cost effective server type, are likely to generalize. Thus, effective repertoire construction requires:

1) a diverse set of previously-produced plans, constructed in response to a wide variety of potential system changes, and
2) a way to consolidate and identify the most plan components that hold the most promise for future reusability

For (1), we build on the idea of *chaos engineering* to explore the space of possible changes by randomly generating change scenarios to generate a diverse base of planning knowledge; we explain in more detail in Section III-A. For (2), we make the observation that plans are, effectively, small programs, and our goal in analyzing them is to identify semantically-meaningful programs or program pieces that may be informative for future use. We thus propose two techniques for this analysis phase, one that adapts clone detection to this domain (Section III-B), and another that proposes a set of rule-based plan transforms to identify cost-effective plan pieces (Section III-C).

Attribute Type	Selection Rate
Utility Coefficients	13.33%
Tactic Failure Rates	23.33%
Number of Users	15.75%
Instance Cost	15.75%
Instance Power	15.75%
Instance Brownout	15.75%

TABLE II: Scenario attribute type and selection probability during mutation.

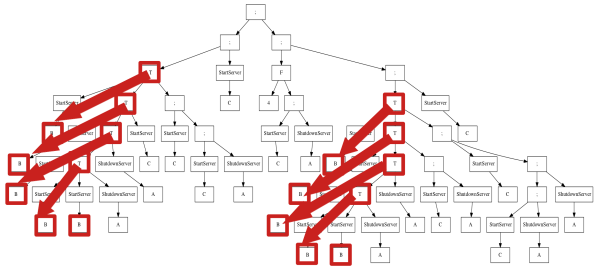

Fig. 3: An example of a clone within a plan.

A. Generating Unexpected Changes

Our technique requires a diverse set of starting strategies that may generalize to future situations. To obtain these strategies, we explore the space of unexpected changes by generating change scenarios using a mutation-based approach inspired by *chaos engineering*. Chaos engineering is an approach to promote software quality attributes such as availability and robustness in large complex systems [8]. It involves subjecting the target system to chaos experiments, which should be conditions that may result in system entering an undesirable state, with the goal of verifying that the system appropriately responds to the experiment. If the system does not respond in an acceptable way, then it can be improved to be more robust to similar situations that might be encountered in production. An example of chaos engineering is Netflix's Simian Army [14].

We therefore propose an approach for building a reusable repertoire by performing chaos experiments offline to obtain a diverse set of adaptation strategies for later reuse. At a high level, this approach randomly selects a scenario attribute, and then randomly mutates it. Because the vast majority of attributes (150 out of 159) are the availability zone specific parameters, random attribute selection is biased to favor the other attributes, to promote scenario diversity. Table II shows this distribution. Attributes within the same type are chosen uniformly at random. Since different attributes have different sensitivity to change, the particular mutation applied depends on the attribute selected. This mutation procedure is repeated m times, where m is the number of desired mutations.

B. Clone detection

Our first intuition for how to improve a repertoire constructed from a diverse set of plans is that some planning motifs are more likely to generalize to unexpected situations.

For example, more servers of the most efficient type (the best performance per cost) is useful in a variety of situations, e.g., if the number of users increases or if the processing resources per request increases. Of course, there are other changes where this tactic is not helpful (such as when the quality requirements change dramatically), but overall this applies to many change scenarios. This motif may therefore appear in many of the diverse plans generated in the first phase.

Thus, our first approach leverages *clone detection* to identify reusable plan components that appear in many plans in the scenario set. Clone detection analyzes software for duplicate source code (see refs. [15], [16] for surveys), which aids developers in refactoring code to promote maintainability or eliminate technical debt. Although this technique is more commonly applied with the aim of reducing redundancy, we observe that the idea can identify planning components that are more likely to be generalizable. Figure 3 shows an example of a clone within an adaptation plan. In this plan, a subplan is repeated. Because this clone is duplicated, it possibly contains important planning knowledge; this key knowledge may be more likely to generalize. By extracting just the clone rather than repeating the full plan(s) in the repertoire, the planner can reuse this prior knowledge more cost-effectively. We therefore apply clone detection to the generated adaptation strategies to find those adaptation strategy components that occurred multiple times throughout the considered change scenarios.

Implementation. Our implementation builds on the Deckard [17] clone detection tool. Deckard performs clone detection by encoding abstract syntax tree (AST) subtrees as vectors, and computing the distance between these vectors to identify similar code regions using clustering.

Note that our approach can generalize to any clone detection mechanism. We use Deckard because it operates on generic tree structures (and can thus be straightforwardly adapted to our plan representation), it considers semantics, is scalable to large AST sizes, and has a publicly-available implementation.

We must make changes to the vector generation step to effectively adapt Deckard's approach to our planning context. Converting an AST into numerical vectors produces a representation amenable to clustering; Deckard generates vectors for AST subtrees based on the number and type of child nodes. By default, Deckard does not consider variable identifier names during vector generation. This is sensible for analyzing large programs written in a general purpose language like Java, where identifiers often vary between clones and where the large number of identifiers quickly explodes vector size. However, our planning language is simple by comparison. More importantly, tactic names (like `StartServer` encode considerable semantically meaningful information. We therefore developed our own vector generator step for the planning language that tracks the occurrence of tactic names.

C. Rule-based Plan Transformation

The clone detection approach can automatically identify reusable repeated planning components. However, human domain expertise, particularly in the peculiarities of the planning language and domain, provides an important avenue for further improvement to repertoire construction. Naive human replanning is time-intensive and expensive, and so any mechanism for incorporating expert knowledge into planning must be sensitive to this cost.

We therefore propose a second approach to repertoire improvement based on human-provided, rule-based source-level transformation templates. Such templates are useful for improving general software quality [18], suggesting that transformation templates for our program-like adaptation strategies could usefully improve their quality, in terms of their generalizability and reusability. For example, we can exploit a priori knowledge of our plan grammar and operator semantics to apply plan transformations that avoid generation of redundant or known-expensive subplans.

We use `Comby` for declaratively specifying templates [19]. `Comby` performs transformations on trees using declarative templates that are syntactically close to the underlying programming language; this is our planning language, in this context. Such templates are therefore lightweight and relatively easy-to-write, easing the burden of manually specifying transformation templates. `Comby` generically supports language syntax with little or no configuration, and is thus a suitable tool for generalizing our template-driven approach to other planning languages like `Stitch` [5] or `PRISM` [20].

Transformation rules for plan reuse. Table III summarizes the eight transformation rules we produced for plans in our exemplar system. Each rule reduces the size of the plan by removing subexpressions, corresponding to subplans. To illustrate, consider the first rule provided in Table III. The `seq-take-first` rule matches a sequence expression (denoted by `;`) and binds named identifiers `1` and `2` to its two respective subexpressions. The `:[ ]` syntax denotes a structural *hole* that binds to expressions. The transformation, denoted by $\Rightarrow$ reduces the sequence expression to only the first subexpression, corresponding to identifier `1`.

All syntax besides hole syntax refers to *concrete* syntax in the underlying language, including operator keywords like `T` or `F` and parentheses. `Comby` rules always match balanced parentheses, which ensures that both matched and transformed subexpressions and plans are syntactically well-formed. `Comby` is thus well-suited to transforming expressions corresponding to subtrees (like balanced parentheses), corresponding to subplans. These transformations are generally not expressible using regular expressions and would be otherwise difficult to implement programmatically.[1]

Our rules are informed by the grammar in Figure 1: for each nonterminal operator (i.e., Sequence, Try-catch, and For loop) we wrote a rule that extracts a respective subexpression (`seq-take-*`, `try-take-*` rules), or reduces the number of iterations that subexpressions are evaluated (`try-unnest`, `for-*` rules). In particular, the `seq-take-first` and `seq-take-second` rules pick the first

[1]Applying a rule to expressions in a plan requires a simple command-line invocation: `comby '(T (:[1]) (:[2]) (:[3]))' '(:[1])' plan.ast`

`seq-take-first`	`(; (:[1]) (:[2]))`	⇒	`(:[1])`
`seq-take-second`	`(; (:[1]) (:[2]))`	⇒	`(:[2])`
`try-take-first`	`(T (:[1]) (:[2]) (:[3]))`	⇒	`(:[1])`
`try-take-second`	`(T (:[1]) (:[2]) (:[3]))`	⇒	`(:[2])`
`try-take-third`	`(T (:[1]) (:[2]) (:[3]))`	⇒	`(:[3])`
`try-unnest`	`(T (:[1]) (T (:[1]) (:[2]) (:[3])) (:[3]))`	⇒	`(T (:[1]) (:[2]) (:[3]))`
`for-prune`	`(F i:[1] (:[2]))`	⇒	`(:[2])`
`for-decr`†	`(F i:[1] (:[2]))`	⇒	`(F i:[1] (:[2]))`

TABLE III: Syntax transformation rules for pruning plans. Hole syntax, like `:[1]`, binds an identifier `1` to an expression. Each rule either replaces a nonterminal expression with a subexpression, or reduces the number of times a subexpression is evaluated. †The `for-decr` rule decrements the loop iterator matched by `:[1]` within the fixed integer range 3–10. For brevity, we elide the rewrite rule that decrements these values.

(resp., second) expression from a sequence expression. The `try-take-*` rules pick one of three Try subexpressions. The `try-unnest` rule prunes Try expressions that share identical child nodes in the first and third arguments.[2] The intuition is that structurally similar subtrees can yield similar benefits, and nested repetitions imply duplicative evaluation unlikely to improve performance. Similarly, `for-prune` and `for-decr` reduce the number of times a For loop executes.

Our experience is that writing programs (e.g., in Java) for transformation rules inside the genetic planner is possible but disadvantageous. Transformations expressed in code are less readable, and can contribute to a planner becoming a black-box. Declarative rules easily express lightweight transformations, and decouples the rule-based system from probabilistic plan discovery, offering greater flexibility.

Rule application. We apply the eight rules to the initial repertoire, selectively removing expressions, which results in smaller plans overall. The general intuition is that smaller plans lead to quicker evaluation times, while retaining particularly valuable subplans for reuse, and thus contribute to greater overall utility. The genetic programming planner explores coarse-grained changes (both adding or deleting subplans), with the overall effect of performing additive changes that create ever-larger plans. Thus, it may miss the opportunity to prune less useful subplans (especially those containing large subexpressions), akin to getting stuck in local optima.

IV. Results

In this section we evaluate the approach for generating and reusing repertoires of adaptation tactics for more effective planning in response to unexpected changes described in Section III. The evaluation is based on a simulated self-* system described in Section II-B. We evaluate the following three research questions:

1) Does reusing a repertoire of adaptation strategies in a GP planner result in more effective plan reuse compared to reusing a single adaptation tactic?
2) Can clone detection identify more reusable adaptation strategy components?
3) Can syntactic transforms improve the reuseability of adaptation repertoires?

[2]When the same hole identifiers are used in a rule, the expressions must be syntactically equal for the rule to match.

We ran experiments on an Ubuntu 16.04.6 LTS server with OpenJDK version 1.8.0_242, an Intel Xeon CPU E5-2699 v3 with 72 cores running at 2.30GHz, and 126 GB of RAM. We restricted experiments to 30 cores and 5 GB of memory.

A. The Repertoire

First, we ask whether a basic repertoire of adaptation strategies generated using a chaos engineering approach results in improved planning in response to an unexpected change.

a) Experimental Setup: We performed replanning on the simulated system for 30 randomly generated unexpected change scenarios. We report the utility obtained for replanning based on using (1) the generated repertoire of adaptation tactics, (2) a single plan (as in prior work [7]), and (3) from scratch (no reuse). We generated the unexpected change scenarios by creating 10 scenarios for each of 3 different different settings for the m number of mutations parameter, 1, 5, and 10. This permits exploring how the size of the change influences replanning effectiveness for the approaches.

The repertoire comprises 200 adaptation strategies that we generated for 200 change scenarios. We generated the change scenarios by applying 1–5 random mutations to the baseline scenario (with the number of mutations selected uniformly at random). When replanning using a single adaptation strategy only, we selected the starting adaptation strategy for reuse randomly from the set of 200 adaptation strategies. When replanning from scratch, the population is initialized completely randomly. To generate the starting population from the repertoire, 10% of the population is selected randomly from the repertoire, and the remaining 90% is generated from scratch; these values were taken from the prior work [7].

For all approaches, the genetic program was configured to plan for 30 generations using a population size of 1000. Planning was automatically terminated at 2000 seconds.

b) Results: Figure 4 shows the results for the first 60 seconds of planning. For space constraints, only trials with 10 mutations (the highest and most challenging setting) are shown; the results for 1 and 5 were similar. The vertical axis is the utility obtained by the planner, and the horizontal axis is the planning time in seconds. The graph therefore shows the utility that the system would obtain by executing the best available plan produced by that planning approach at that time. Results from replanning using a single plan are labeled single.

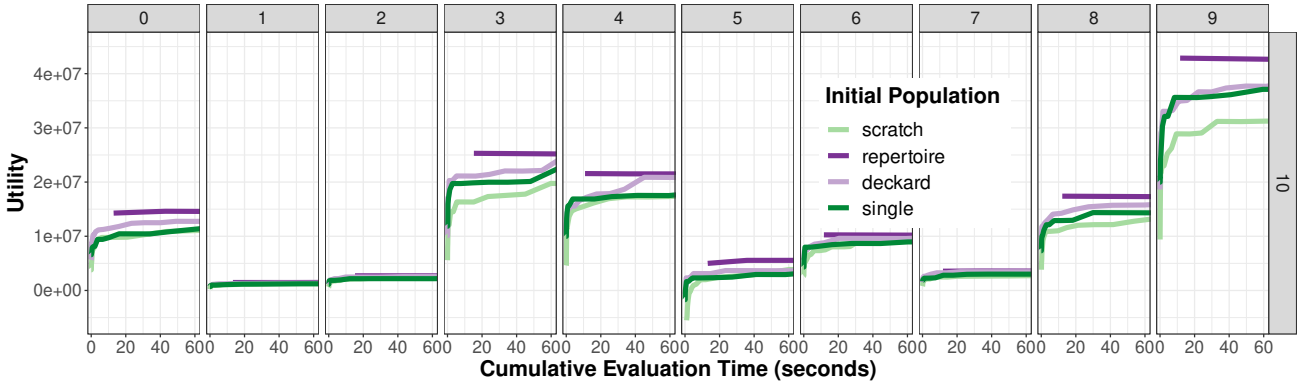

Fig. 4: Results comparing planning from scratch, the repertoire, replanning from a single plan only, and replanning using Deckard. Deckard resulted in better utility for the first 13 seconds of planning, and is then overtaken by the repertoire.

For almost all randomly generated scenarios, the repertoire approach results in the highest planning utility. Sometimes the improvement compared to the next best planner was small, especially for single-mutation cases. For other scenarios the improvement was quite large (such as trial 9 in Figure 4. On aggregate, using the repertoire resulted in an average improvement of 11% to utility compared to reusing a single plan only. Planning using a single plan tends to only outperform planning from scratch, often slightly, reinforcing previous results [7]. One drawback to the repertoire approach is that it takes more time to produce the first plan (often taking around 15 seconds), although the plan that is obtained is often high quality. This is intuitive since effective plans are often large and expensive to evaluate, and the repertoire approach must evaluate many of these large and expensive plans. If planning in a domain where waiting 15 seconds is unacceptable, then reusing a single plan is better. Otherwise, the repertoire results in the highest expected utility.

B. Clone Detection

Next, we ask whether initializing the population from clones is an effective strategy for identifying reusable planning components. To answer this question, we performed replanning on the same randomly generated unexpected change scenarios as in Section IV-A using a clone detection approach to initialize the population. To do this, we ran Deckard on the repertoire of 200 adaptation strategies generated in the previous subsection to obtain a list of clones. Clones were selected from this list using tournament selection, selecting seven clusters randomly from the list and returning a random clone from the largest cluster. The initial population was initialized with these clones. The result of this strategy is shown in Figure 4, labeled as deckard.

Overall, the clone detection approach results in an improvement compared to planning from scratch and replanning with a single plan only, but the maximum utility was obtained by reusing the repertoire rather than the extracted clones. Nevertheless, the clone detection approach yields plans more quickly. Given enough planning time, the repertoire approach eventually finds a better plan than the clone detection approach, but when a small amount of time is available, the clone detection approach is better. The breakeven point, where both clone detection and the repertoire are best for an equal number of the trials, occurs after 13 seconds of planning. For the first 10 seconds of planning, the clone detection approach yields the highest utility for 24 out of the 30 trials, with reusing a single plan being the best for 4 trials and planning from scratch the best for the remaining 2 trials. When planning for longer than 13 seconds the repertoire approach is expected to result in the highest utility.

C. Rule-based Syntactic Transforms

The third research question addresses the efficacy of tailoring reusable plan fragments with syntactic transformations. For this experiment we generated adaptation strategies for the same 30 unexpected change scenarios as before, while applying eight syntactic transformations (as described in in Section III-C). For each syntactic transformation, we applied the transformation to the starting repertoire of 200 adaptation strategies from prior experiments, and then used the transformed repertoire to seed the initial population for replanning. The large number of trials makes the results of this experiment difficult to show visually, so results are shown in Table IV.

Of the eight transforms evaluated, four result in an improvement over the baseline (the repertoire with no transforms) more than half of the time. The `try-take-first` performed the best, improving on the baseline for 29/30 trials, and with an average improvement to expected utility of 3.51%. This improvement is consistent across each of the three numbers of mutations in the experiment. The `for-prune` transformation also results in an improvement for all three numbers of mutations, but with a lower percentage of trials improved (63.3%) and a lower overall improvement to utility (0.56%). The other two transforms that showed an overall improvement were `try-take-third` and `try-unnest`. These transforms both improved about 60% of trials for 0.46% and 0.26% average improvement respectively. The other transforms resulted in an overall decrease to expected utility.

Rule	Trials Improved (%)	Overall % Change	1 Mutation % Change	5 Mutation % Change	10 Mutation % Change
`seq-take-first`	40.0	-0.36	0.57	-0.93	-0.53
`seq-take-second`	26.7	-2.13	-2.36	-0.89	-3.01
`try-take-first`	96.7	3.51	3.79	3.27	3.53
`try-take-second`	36.7	-0.75	-2.36	-0.89	-1.56
`try-take-third`	63.3	0.46	0.89	-0.06	0.59
`for-decr`	40.0	-0.43	0.93	-0.26	-1.52
`for-prune`	63.3	0.56	0.58	0.71	0.42
`try-unnest`	60.0	0.26	0.51	-0.13	0.40

TABLE IV: Improvement in maximum utility obtained by syntactic transforms over using the repertoire without transforms. `try-take-first` performed the best with a consistent 3.5% improvement.

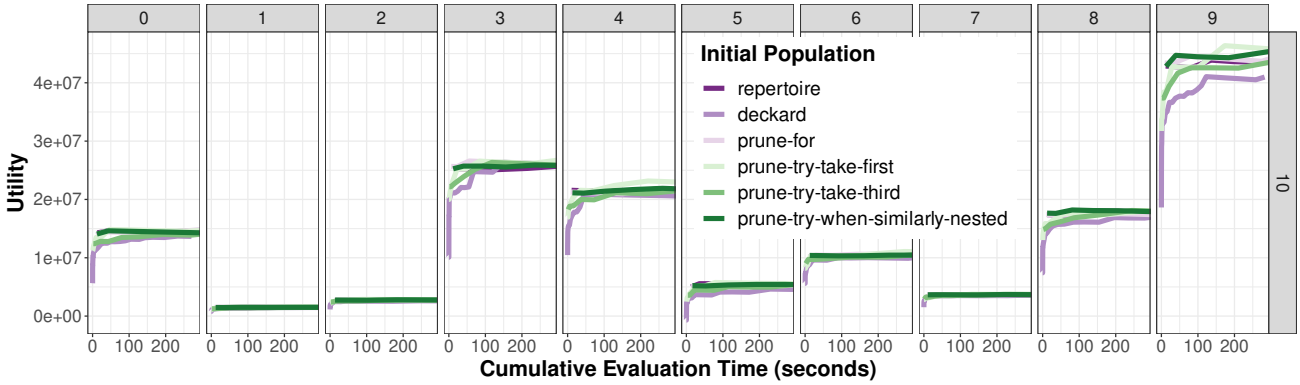

Fig. 5: Utility versus planning time for the four beneficial syntactic transforms. Some transforms obtained results as quick as Deckard but with better utility. `try-take-first` is the overall best after around 2 minutes of planning.

The biggest takeaway from these results is the strong performance of the `try-take-first` transform, which improves utility for all but one trial for 3.51% on average. Compared to replanning with a single plan only, this transform resulted in an overall average improvement of 15%, with the best overall improvement being 20% for trial 9. It is interesting that `try-take-first` performed much better than other `try-*` templates, since all these transforms similarly prune subtrees of Try-catch operators. The difference is that the `try-take-first` picks the first subtree of the Try-body and removes other subtrees, while remaining templates remove other parts of the Try-catch subtrees. This result makes sense intuitively since the transform captures what is likely the most important information contained in the Try-catch operator (the subplan that is attempted first) while reducing evaluation time on evaluating the contingencies. We've previously noted a common planning motif where an important subplan is tried multiple times to ensure that it is carried out, should it fail a few times. Crucially, when this occurs, capturing the important subplan with a tailored rule allows the planner to reuse the information learned during subsequent replanning while reducing the evaluation time.

Figure 5 shows the results of the syntactic transforms versus clone detection (deckard), and using the repertoire without modifications (repertoire). For presentation, we show only the four transforms that result in a positive average improvement. Compared to planning from scratch or reusing a single plan (shown as single), replanning using only the repertoire results in the highest utility, but typically requires more time to begin returning results (cf. Section IV-A). Overall, Table IV shows that the `try-take-first` improves on the repertoire by 3.5%. Figure 5 shows that syntactic transforms generate plans as quickly as the Deckard-based planner, but with consistently higher utility. Interestingly, the `try-unnest` and `for-prune` transforms take about as long to start returning plans as the repertoire alone, around 15 seconds, but result in higher utility than `try-take-first` for the first minute or two of planning. If a plan is needed within a short time window, such as 10 seconds, `try-take-first` is the best approach. For an intermediate window between around 15 seconds to 60 seconds, `try-when-similarly-nested` or `for-prune` perform the best. When planning longer than 60 seconds is permissible, `try-take-first` is again most effective.

D. Discussion

Generating a repertoire of adaptation strategies by exploring the change space on its own resulted in an overall improvement to utility of 11%. The clone detection approach resulted in better quality adaptation strategies for the first 10 seconds of planning compared to the repertoire on its own, but when planning for longer than 10 seconds reusing the repertoire on its own was better. The best results were obtained by the `try-take-first` syntactic transformation, which resulted in a 3.5% improvement over the repertoire on its own, although

this approach required one to two minutes of planning before obtaining the best utility. These results show that our approaches for building reusable repertoires can enable self-* systems to more effectively apply existing knowledge to replan following an unexpected change.

There are a number of threats to the generality of the results that we attempted to minimize, such as by evaluating on an indicative self-* system inspired by AWS. Once key concern is that the kinds of unexpected changes generated during the evaluation may be different from the kinds of changes that a self-* system may actually need to replan for. Indeed, by their nature, we cannot expect the kinds of unexpected changes that will arise. This means that the results cannot be taken to show that the approach will always result in a 15% improvement. Rather, they demonstrate how it is possible to apply existing planning knowledge from the situations that were expected at design time, to at least some types of unexpected changes that are similar in nature to those kinds of changes that were considered. This is shown by the results from different numbers of mutation operations used to generate the change scenarios, particularly when replanning for change scenarios generated with 10 mutations, twice the number of allowed mutations that could be considered during the offline phase, meaning these scenarios were fully "unexpected" from the point of view of the system during the offline phase. Despite this, plan reuse still resulted in an improvement for these experiments (for the repertoire, 12% average improvement for the scenarios with 10 mutations compared to 11% average improvement overall). Of course, the amount of improvement is likely to be less for 1000+ mutations or for mutation operators dramatically different from those used offline, but as we cannot realistically expect to plan for every possible unknown unknown, we instead seek to reuse as much existing planning knowledge as possible.

Another issue related to generality is how well the approach will scale to more complicated planning problems. An advantage of approaches using stochastic search is that these approaches scale well to large search spaces. However, stochastic search methods are sensitive to the shape of the search space, and are likely to struggle when the search space is difficult to explore. Additionally, the planning knowledge contained in other planning problems may be more or less reusable. Regarding the applicability of the repertoire construction approaches to other problems, the clone detection approach can be readily applied to other planning problems with an AST representation. The specific syntactic transformations used in this work will likely need to be modified to work for other problems, although some ideas like removing for loops may be transferable, at a minimum the syntax may need to be updated to reflect the particularities of the planning language. On the other hand, the flexibility provided by the syntactic transformation approach allows for experts to fine tune the approach to the demands of other more challenging problems. It is also possible to combine the three approaches (repertoire, clone detection, and transforms) and we leave investigating these issues to future work.

V. Related Work

Self-* planning and reuse. There are many approaches for self-adaptive systems planning, including model-based approaches [4], manual human written plans [5], and heuristic planners such as genetic algorithms [7], [21]–[23]. In this work, we focus on planners built around evolutionary algorithms that reuse prior effective adaptation strategies [7], [24], [25], in particular building on our prior work [7] that showed how reusing single plans encoded as ASTs can allow a self-adaptive system to replan more effectively following an unexpected change. In this work, we investigate reusing a repertoire of adaptation strategies, and explore two approaches for identifying reusable planning components for constructing repertoires that are amenable to reuse.

EvoChecker [24] is an evolutionary approach for generating generating probabilistic models, and can be used to reconfigure self-* systems at runtime. EvoChecker supports reusing prior solutions by seeding the search with archived prior solutions, and three update rules for maintaining this archive were studied, including storing the complete population from the current adaptation step, storing the best two configurations from the current adaptation step, and accumulating the previous best two configurations from all previous adaptation steps. Our approach for repertoire construction instead proactively builds a repertoire of effective responses for potential unexpected situations that may arise in the future.

Seeding an evolutionary algorithm has also been studied in the area of service-oriented computing [25], where a multi-objective evolutionary algorithm can reuse previously stored service composition plans. This work investigates four seeding strategies, including strategies that are pregenerated for the purpose of obtaining knowledge about the problem (like our approach), as well as two strategies that reuse previously solved composition problems. In this work, we build repertoires of solutions encoded as ASTs rather than service composition plans. Additionally, we present approaches for extracting the reusable portions of prior solutions to improve reuse effectiveness.

Case-based plan adaptation [26] also reuses previous planning solutions in new contexts, including maintaining repertoires of previous solutions. Evolutionary algorithms have been explored in this context [27], [28], e.g., by injecting solutions to previous problems into a GA population to speed the solution of new problems. Although the seeding approach to reuse is similar, these works do not address the challenge of reusing previous solutions in response to unexpected changes in self-* systems.

Program analysis. We build two repertoire customization approaches based on techniques from clone detection and rule-based syntax transformation. There are many approaches for performing clone detection in the software analysis and maintenance literature (see ref. [15], [16] for surveys). In this work, we use the well known Deckard [17] clone detection tool to find clones in self-* adaptation strategies. While much of the clone detection in software engineering literature seeks

to avoid clones as indicative of technical debt, we instead promote clones as signaling generalizable features.

There is also a large body of work on declarative syntax transformation techniques (e.g., [29], [30]). We use `Comby` [19], [31] particularly because it operates on our planning language without any additional special configuration and delivers fast, lightweight, readable transformation templates. Alternative tools would require us to write a parser or grammar specification that integrates with an existing framework, or require more effort to write transformation rules. To the best of our knowledge, our approach is the first to implement and evaluate a rule-based transformation approach in conjunction with genetic programming for building reusable self-* plans.

VI. Conclusion

While plan reuse with stochastic search is a promising idea for enabling self-* systems to replan following an unexpected change, reusing repertoires of adaptation strategies presents new challenges for plan reuse in self-* systems. In this work, we present a two part approach for constructing reusable repertoires of adaptation strategies that are likely to generalize and are cost effective to evaluate. This approach first takes inspiration from chaos engineering to obtain diverse set of adaptation strategies, and then applies ideas from program analysis to identify planning pieces amenable to reuse. We investigate two approaches, clone detection and syntactic transformations, for this analysis step, and evaluate our approach on a simulated self-* system inspired by AWS. The results found that the most effective approaches resulted in an improvement over prior work by up to 20%. Proactively building reusable repertoires of adaptation strategies is a step towards self-* systems robust to a wide range of unexpected changes.

References

[1] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-based self-adaptation with reusable infrastructure," *Computer*, vol. 37, no. 10, pp. 46–54, 2004.

[2] C. Krupitzer, F. M. Roth, S. VanSyckel, G. Schiele, and C. Becker, "A survey on engineering approaches for self-adaptive systems," *Pervasive and Mobile Computing*, vol. 17, pp. 184–206, 2015.

[3] J. Zhang and B. H. Cheng, "Model-based development of dynamically adaptive software," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 371–380.

[4] A. Pandey, G. A. Moreno, J. Cámara, and D. Garlan, "Hybrid planning for decision making in self-adaptive systems," in *2016 IEEE 10th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*. IEEE, 2016, pp. 130–139.

[5] S.-W. Cheng and D. Garlan, "Stitch: A language for architecture-based self-adaptation," *J. Syst. Softw.*, vol. 85, no. 12, pp. 2860–2875, 2012.

[6] B. Nebel and J. Koehler, "Plan reuse versus plan generation: A theoretical and empirical analysis," *Artificial Intelligence*, vol. 76, no. 1-2, pp. 427–454, 1995.

[7] C. Kinneer, Z. Coker, J. Wang, D. Garlan, and C. Le Goues, "Managing uncertainty in self-adaptive systems with plan reuse and stochastic search," in *International Conference on Software Engineering for Adaptive and Self-Managing Systems*, 2018, pp. 40–50.

[8] A. Basiri, N. Behnam, R. De Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal, "Chaos engineering," *IEEE Software*, vol. 33, no. 3, pp. 35–41, 2016.

[9] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[10] R. Poli, W. B. Langdon, N. F. McPhee, and J. R. Koza, *A field guide to genetic programming*. Lulu.com, 2008.

[11] J. H. Holland, "Genetic algorithms," *Scientific american*, vol. 267, no. 1, pp. 66–73, 1992.

[12] G. A. Moreno, B. Schmerl, and D. Garlan, "Swim: an exemplar for evaluation and comparison of self-adaptation approaches for web applications," in *Proceedings of the 13th International Conference on Software Engineering for Adaptive and Self-Managing Systems*. ACM, 2018, pp. 137–143.

[13] "Choose your aws region wisely," https://www.concurrencylabs.com/blog/choose-your-aws-region-wisely/, Concurrency Labs, accessed: 2020-02-18.

[14] Y. Izrailevsky and A. Tseitlin, "The netflix simian army," https://netflixtechblog.com/the-netflix-simian-army-16e57fbab116, The Netflix Tech Blog, accessed: 2020-3-23.

[15] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *Queen's School of Computing TR*, vol. 541, no. 115, pp. 64–68, 2007.

[16] A. Sheneamer and J. Kalita, "A survey of software clone detection techniques," *International Journal of Computer Applications*, vol. 137, no. 10, pp. 1–21, 2016.

[17] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007, pp. 96–105.

[18] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic Patch Generation Learned from Human-written Patches," in *International Conference on Software Engineering*, ser. ICSE '13, 2013, pp. 802–811.

[19] "Comby," https://comby.dev, Online, accessed 13 May 2020.

[20] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of probabilistic real-time systems," in *Int. Conf. on Computer Aided Verification*, ser. CAV '11, 2011, pp. 585–591.

[21] E. M. Fredericks, I. Gerostathopoulos, C. Krupitzer, and T. Vogel, "Planning as optimization: Dynamically discovering optimal configurations for runtime situations," in *International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*. IEEE, 2019, pp. 1–10.

[22] A. J. Ramirez, B. H. Cheng, P. K. McKinley, and B. E. Beckmann, "Automatically generating adaptive logic to balance non-functional tradeoffs during reconfiguration," in *Int. Conf. on Autonomic Computing*, ser. ICAC, 2010, pp. 225–234.

[23] T. Chen, K. Li, R. Bahsoon, and X. Yao, "FEMOSAA: feature guided and knee driven multi-objective optimization for self-adaptive software at runtime," *CoRR*, vol. abs/1608.08933, 2016. [Online]. Available: http://arxiv.org/abs/1608.08933

[24] S. Gerasimou, R. Calinescu, and G. Tamburrelli, "Synthesis of probabilistic models for quality-of-service software engineering," *Automated Software Engineering*, vol. 25, no. 4, pp. 785–831, 2018.

[25] T. Chen, M. Li, and X. Yao, "Standing on the shoulders of giants: Seeding search-based multi-objective optimization with prior knowledge for software service composition," *Information and Software Technology*, vol. 114, pp. 155–175, 2019.

[26] H. Muñoz-Avila and M. T. Cox, "Case-based plan adaptation: An analysis and review," *IEEE Intelligent Syst.*, vol. 23, no. 4, pp. 75–81, 2008.

[27] A. Grech and J. Main, *Case-Base Injection Schemes to Case Adaptation Using Genetic Algorithms*, ser. ECCBR, Berlin, Heidelberg, 2004, pp. 198–210.

[28] S. J. Louis and J. McDonnell, "Learning with case-injected genetic algorithms," *Trans. Evol. Comp*, vol. 8, no. 4, pp. 316–328, 2004.

[29] L. Wasserman, "Scalable, example-based refactorings with refaster," in *Workshop on Refactoring Tools*, ser. WRT@SPLASH '13, 2013, pp. 25–28.

[30] J. I. Maletic and M. L. Collard, "Exploration, analysis, and manipulation of source code using srcml," in *International Conference on Software Engineering*, ser. ICSE '15, 2015, pp. 951–952.

[31] R. van Tonder and C. Le Goues, "Lightweight Multi-Language Syntax Transformation with Parser Parser Combinators," in *Programming Language Design and Implementation*, ser. PLDI '19, 2019, pp. 363–378.